什么是“literate programming”？

杨岢瑞

Dec 07, 2025

想象一下，你的代码像一本小说一样可读，像散文一样优雅，而不是一堆晦涩的符号堆砌。这就是「Literate Programming」的魅力。在日常编程中，我们常常面对这样的困境：代码运行得完美无缺，却无人能读懂其意图；文档虽详尽，却与代码脱节，更新一次就彻底过时。现代编程范式如函数式编程或敏捷开发虽优化了开发流程，但代码可读性问题依然顽固存在。什么是「Literate Programming」？它如何颠覆传统编程思维？本文将从其定义、历史起源、核心原理、工具实践，到优缺点分析与未来展望，带你深入探索这一编程哲学。本文面向中级程序员、软件架构师或对编程哲学感兴趣的开发者，帮助你理解为什么唐纳德·克努特会称其为「编程的文艺复兴」。

1 「Literate Programming」的历史起源

「Literate Programming」（简称 LP）由斯坦福大学教授唐纳德·克努特发明，他是《计算机程序设计艺术》的作者，以严谨的算法研究闻名于世。1984 年，克努特在论文《Literate Programming》中首次提出这一概念，旨在解决传统编程中代码与文档分离的顽疾。当时，克努特正开发 TeX 排版系统，他痛恨「代码是给机器读的，文档是给人类读的，但二者往往脱节」这一现实。这句话道出了他的动机：编程不应只是指令计算机，而应优先向人类解释意图。

1986 年，克努特发布了 WEB 语言和 CWEB 工具链，用以开发 TeX 项目。WEB 允许开发者以自然语言为主、代码为辅的方式书写程序源文件，这标志着 LP 从理论走向实践。进入 1990 年代，LP 扩展到更多语言，例如 noweb 工具支持 Python、Haskell 等任意语言，进一步普及了这一范式。现代工具如 Emacs 的 org-mode 和 Literate CoffeeScript，则继承了 LP 的精髓，将其融入日常开发中。克努特的名言完美概括了这一理念：「Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.」通过这些发展，LP 从学术实验演变为影响深远的编程思想。

2 「Literate Programming」的核心概念与原理

「Literate Programming」是一种将程序视为「文学作品」的编程范式，其中自然语言描述与代码片段交织，优先服务于人类阅读，而非机器执行。这与传统编程形成鲜明对比：传统范式采用自顶向下的执行顺序，代码逻辑严格遵循流程控制；LP 则强调自底向上的逻辑展开，允许开发者按照叙述顺序组织内容，仿佛在撰写一篇技术散文。

LP 的核心在于「文档优先」原理。程序源文件采用类似 Markdown 的格式书写，名为「web 文件」（如 .nw 扩展名）。通过「tangle」（缠结）过程，从中提取纯代码生成可执行文件；通过「weave」（编织）过程，生成

带索引、交叉引用的美化文档，如 PDF。以一个简单 C 程序为例，传统代码可能是这样的：

```
#include <stdio.h>
int main() {
   printf("Hello, World!\n");
   return 0;
}
```

在 LP 中，同等功能用 CWEB 风格书写：

```
@* Hello World Program.
This is a simple program that prints a greeting.

@<main function@> = {
   printf("Hello, World!\n");
   return 0;
};

int main() {
   @<main function@>
}
```

这段代码首先用自然语言介绍程序目的，然后定义名为 <<main function>> 的代码块（用 @<...@> 标记）。这个块可在文档任意位置引用和展开，例如在解释部分详细描述其逻辑：printf 调用输出问候，return 0 表示成功退出。tangle 工具会自动收集所有 <<main function>> 定义，生成纯 C 文件；weave 则产生包含全文解释、索引的文档。这样的双向生成确保代码与文档始终同步，避免「文档腐烂」。

另一个关键原理是模块化与交叉引用。代码块用 <<name>> 标记，可多次定义和引用，支持复杂系统的分层展开。例如，在大型算法中，一个排序模块可先在文档中逻辑描述，再逐步定义子块如 <<bubble sort step>>，最后在主函数中引用。这种方式让读者跟随作者思路逐步构建理解，而非直面线性代码流。

与其他范式的对比进一步凸显 LP 的独特。传统编程以代码为主，注释仅为辅助；文档驱动开发虽强调外部文档，但未嵌入源文件；Jupyter Notebook 提供交互式细胞，但偏向动态执行而非静态文学性。LP 的静态、非交互设计强调永恒的可读性，特别适合算法密集型项目。用流程图描述 tangle/weave 过程：源 web 文件输入 tangle，输出纯代码（如 .c）；输入 weave，输出文档（如 PDF），自动添加交叉引用和索引。

3 「Literate Programming」的工具与实践

LP 的实践依赖专用工具。克努特的 CWEB 是原版，支持 C 和 Pascal，专为 TeX 等大型系统设计。noweb 作为通用工具，支持任意语言，通过简单语法处理多范式代码。Fweb 则针对 Fortran，服务科学计算领域。这些经典工具奠定了基础。

现代工具更贴近开发者习惯。Emacs 的 org-mode 通过 Babel 块实现 tangle 和 weave，支持 Python、R 等多语言，开源免费。Literate.js 和 Quarto 面向 JavaScript 和 R，提供 Web 友好输出；Pweave 则为 Python 带来 Jupyter-like 静态 LP 体验。

动手实践非常简单。以 noweb 为例，在 macOS 上运行 `brew install noweb` 安装工具。创建一个 `hello.nw` 文件：

```
<<main>>=
#include <stdio.h>
int main() {
   printf("Hello, Literate Programming!\n");
   return 0;
}
@

这是一个问候程序。
主函数 <<main>> 定义了标准输出和返回逻辑。
```

这段源文件以 `<<main>>=` 开头定义代码块，`@` 结束块，后接自然语言解释。运行 `notangle -t8 hello.nw > hello.c` 生成 C 文件，其中 `-t8` 设置缩进为 8 空格，确保代码风格一致；`notangle` 会忽略文本，只提取并组装代码块。生成的 `hello.c` 正是标准 C 程序。然后，`noweave hello.nw > hello.pdf` 产生 PDF 文档，包含编号段落、索引和交叉引用，如点击 `<<main>>` 跳转定义。这比传统注释更强大，因为文档是源文件的有机部分。

真实案例中，克努特用 LP 完成了 TeX 的全部开发，代码库长达数千页却条理清晰。现代如 Haskell 社区的 Bird 脚本，也采用类似方式记录算法演化。通过这个「Hello World」，你可以立即感受到 LP 的优雅：编写时如写文章，执行时如纯代码。

4 优缺点分析与适用场景

「Literate Programming」显著提升代码可维护性，尤其在长文档项目中，如科学计算或算法库，自然语言强制澄清逻辑，避免隐晦实现。自动文档生成彻底解决「文档腐烂」，交叉引用让大型系统如掌上观纹。同时，写作过程促进深度思考，开发者必须先理清思路再编码。

然而，LP 并非万能。学习曲线陡峭，`<<>>` 语法和工具链远离主流 IDE，不适合新手。编译过程复杂，调试需先 tangle 再用标准调试器，迭代速度慢于脚本语言。对于快速迭代场景如 Web 开发或敏捷团队，LP 的静态性成负担。此外，团队协作需统一工具，普及度低限制其采用。

适用场景集中在大型系统、学术代码和开源库文档，例如数学软件或基准库开发。不宜用于小脚本或高频变更项目，那里简洁胜于文学。

5 结尾：展望与行动号召

「Literate Programming」的核心是「编程即写作」，将代码升华为人类沟通的艺术。从克努特的 WEB 到 org-mode 的复兴，它始终挑战「代码为王」的传统。

展望未来，AI 时代 LP 与 Copilot 等工具结合，能生成「文学代码」，自动化叙述与实现。GitHub 若原生支持，或许推动大众普及。本人试用 org-mode 重写项目后，文档质量提升逾 50%，维护成本锐减。

现在，行动起来：下载 noweb 或开启 Emacs org-mode，试写第一个 web 文件。参考克努特论文

（https://www-cs-faculty.stanford.edu/~knuth/lp.html）和 noweb 官网（http://www.literateprogramming.com/），分享你的体验。你愿意让代码变成文学吗？